Exercise 1F-2. Language Design [5 points]. Comment on some aspect from Hoare's *Hints On Programming Language Design* that relates to your programming experience. Provide additional evidence in favor of one his points and against one of his points. Do not exceed three paragraphs. Both your ideas and also the clarity with which they are expressed (i.e., your English prose) matter. Readers should be able to identify your main claim, the arguments you are bringing to bear, and your conclusion.

Answer:

One aspect of Hoare's *Hints on Programming Language Design* that relates to my programming experience is the idea of self-modifying code, also known as metaprogramming. As Hoare explains in Section 8 ("Variables"), "One of the most powerful and most dangerous aspects of machine code programming is that each individual instruction of the code can change the content of any register, any location of store, and alter the condition of any peripheral: it can even change its neighboring instructions or itself" [1]. My first exposure to this concept was in EECS 370 at the University of Michigan, where, as part of a project, I wrote assembly instructions that changed and inserted other assembly instructions. I also thought the idea of self-modifying code was strange and dangerous just as Hoare discusses, but it was not until my second software engineering internship that I understood the power of this concept in practice. During my internship, I used the C++ Boost.Preprocessor library to perform preprocessor metaprogramming for a financial technology application. With this library, I wrote a program to generate C++ classes and functions from datasets involving hundreds of financial securities, removing the need to hand-code them all. Hoare is correct in that metaprogramming can have dangerous effects, but it also can enhance the scalability of real-world applications if done correctly.

After his short discussion of machine code metaprogramming and how disjoint variables can mitigate its risks, Hoare makes a questionable claim about references and pointers with which many programmers would disagree. As Hoare describes, "Unlike all other values (integers, strings, arrays, files, etc.) references have no meaning independent of a particular run of a program . . . Their introduction into high level languages has been a step backward from which we may never recover" [1]. Contrary to Hoare's claim, references and pointers are useful in solving some of computer science's most common practical problems. One such problem is performance in function parameter passing. For example, in C++, the default method of function parameter passing is pass-by-value, which means that large data structures like vectors will be deep-copied when passed directly as parameters to a function [2]. Deep copies of large data structures can cost significant time and memory. With references or pointers, on the other hand, these copies can be done in constant time and memory by copying pointers or references instead. Furthermore, pointers can give programmers control over memory management. In C++, pointers can point to any memory location consisting of data of any type, and the programmer can allocate and free memory as needed with the keywords `new` and `delete` [3]. This ability gives programmers freedom to store, decode, and manage memory however they like, which allows them to optimize for specific use cases. Given the advantages of references and pointers, it would not be accurate to call them a "step

backward from which we may never recover," as they are often critical in uses of modern programming languages.

One of Hoare's more favorable claims is that block structure is a powerful tool in the organization of high level programming languages. As Hoare exclaims, "[H]igh level languages provide the programmer with a powerful tool for achieving even greater security, namely the scope and locality associated with block structure" [1]. Many of today's most popular programming languages, such as C++ and Java (and Python to an extent, through indentation), use block structure to denote variable scope [4]. Without block structure, the scoping features and memory efficiency that make these languages popular would not exist. Additionally, block structure makes programming languages easier to understand from an educational standpoint [5], which can encourage newcomers to the field of programming languages to participate and contribute new ideas. Overall, Hoare provides substantial evidence to support each of his arguments in his paper. I found his short discussion of self-modifying code to be interesting, and while his views on references and pointers are questionable, his views on block structure are well-supported by more modern developments in programming languages.

Exercise 1F-3. Simple Operational Semantics [3 points]. Consider the IMP language discussed in class, with the Aexp sub-language extended with a division operator. Explain what changes must be made to the operational semantics (big-step only). Write out formally any new rules of inference you introduce.

Solution:

To add the integer division operator, we must add a new rule of inference that performs the floor division of two arithmetic expressions. We must also specify that the resulting value of the divisor is not equal to 0 (because division by 0 is not mathematically defined). The new rule is below:

$$\frac{\langle e_1, \sigma\rangle \Downarrow n_1 \quad \langle e_2, \sigma\rangle \Downarrow n_2}{\langle e_1 / e_2, \sigma\rangle \Downarrow \lfloor n_1 / n_2 \rfloor} \text{, where } n_2 \neq 0$$

Exercise 1F-4. Language Feature Design, Large Step [10 points]. Consider the IMP language with a new command construct "`let` $x = e$ `in` c". The informal semantics of this construct is that the Aexp e is evaluated and then a new local variable x is created with lexical scope c and initialized with the result of evaluating e. Then the command c is evaluated. We also extend IMP with a new command "`print` e" which evaluates the Aexp e and "displays the result" in some un-modeled manner but is otherwise similar to `skip`.

We expect (the curly braces are syntactic sugar):

```
x := 1 ;
y := 2 ;
{ let x = 3 in
```

```
  print x ;
  print y ;
  x := 4 ;
  y := 5
} ;
print x ;
print y
```

to display "3 2 1 5".

Extend the natural-style operational semantics judgment $\langle c, \sigma \rangle \Downarrow \sigma'$ with one new rule for dealing with the `let` command. Pay careful attention to the scope of the newly declared variable and to changes to other variables.

Solution:

The new rule to handle the `let` command is shown below:

$$\frac{\langle e, \sigma \rangle \Downarrow n_1 \quad \langle x, \sigma \rangle \Downarrow n_2 \quad \langle x := e, \sigma \rangle \Downarrow \sigma[x := n_1] \quad \langle c, \sigma[x := n_1] \rangle \Downarrow \sigma'[x := n_2]}{\langle \texttt{let } x = e \texttt{ in } c, \sigma \rangle \Downarrow \sigma'[x := n_2]}$$

This rule considers 4 evaluations:

- It evaluates e in state σ, producing n_1.
- It evaluates x in state σ, producing n_2.
- It evaluates the assignment $x := e$ in state σ, producing the same state σ but with $x := n_1$ (we label this state as $\sigma[x := n_1]$).
- It evaluates c in state $\sigma[x := n_1]$, producing a new state σ' but with x reset to its original value n_2 (we label this state as $\sigma'[x := n_2]$).

The result is that the `let` command produces a new state $\sigma'[x := n_1]$ where x is set to its original value before the `let` command.

Exercise 1F-5. Language Feature Design, Small Step [10 points]. Extend the set of redexes, contexts and reduction rules for the contextual-style operational semantics that we discussed in class to account for the `let` command introduced above.

Solution:

First, we add a new redex:

$$r ::= \ \ldots \mid \texttt{let } x = e \texttt{ in } c$$

where the "..." represents the other redexes on slide 26 of the Lecture 05 slides. We then add a new local reduction rule:

$$\langle \texttt{let } x = e \texttt{ in } c, \sigma \rangle \to \langle x := e \ ; \ c \ ; \ x := \sigma(x), \sigma \rangle$$

Finally, we do not add any new contexts to the ones discussed in class, as the local reduction of `let` transforms it into a sequence of commands whose contexts are already defined.

Exercise 1C. Language Feature Design, Coding. Download the Homework 1 code pack from the course web page. Modify `hw1.ml` so that it implements a complete interpreter for IMP (including `let` and `print`). Base your interpreter on IMP's large-step operational semantics. The `Makefile` includes a "`make test`" target that you should use (at least) to test your work.

Modify the file `example-imp-command` so that it contains a "tricky" terminating IMP command that can be parsed by our IMP test harness (e.g., "`imp < example-imp-command`" should not yield a parse error).

Submission. Turn in the formal component of the assignment (1F-1 through 1F-5) as a single PDF document via the `gradescope` website. Your name and Michigan email address must appear on the first page of your PDF submission but may not appear anywhere else. Turn in the coding component of the assignment (1C) via the `autograder.io` website.

References

[1] C. A. Hoare, "Hints on Programming Language Design", *Stanford University Dept. of Computer Science*, 1973.

[2] A. Pomeranz, "10.2 - Passing argumets by value", *Learncpp.com*, July 2007. [Online] Available: https://www.learncpp.com/cpp-tutorial/passing-arguments-by-value/

[3] "Dynamic memory", *cplusplus.com*, November 2014. [Online]. Available: https://www.cplusplus.com/doc/tutorial/dynamic/

[4] "Block Structure and Scope", *Weber State University Dept. of Computer Science*, September 2020. [Online]. Available: https://icarus.cs.weber.edu/ dab/cs1410/textbook/3.Control/blocks.html

[5] O. L. Madsen, "Block structure and object oriented languages," in *Proceedings of the 1986 SIGPLAN workshop on Object-oriented programming*, 1986, pp. 133–142.

1 HW1 (select all pages: your first page has your name and bookkeeping, and all others are anonymous))

- **0 pts** Correct